МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ

НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ

«КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ»

Факультет електроенерготехніки та автоматики

МЕТОДИЧНІ ВКАЗІВКИ ТА ЗАВДАННЯ
до виконання практичних робіт з дисципліни

«ОБЧИСЛЮВАЛЬНА ТЕХНІКА ТА ПРОГРАМУВАННЯ. ЧАСТИНА 2»
для студентів спеціальності
141 Електроенергетика, електротехніка та електромеханіка

(навчальне електронне видання)

НТУУ “КПІ”
2016

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ

НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ

«КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ»

Факультет електроенерготехніки та автоматики

МЕТОДИЧНІ ВКАЗІВКИ ТА ЗАВДАННЯ
до виконання практичних робіт з дисципліни

«ОБЧИСЛЮВАЛЬНА ТЕХНІКА ТА ПРОГРАМУВАННЯ. ЧАСТИНА 2»
для студентів спеціальності
141 Електроенергетика, електротехніка та електромеханіка

(навчальне електронне видання)

Рекомендовано Вченою радою
факультету електроенерготехніки та автоматики

НТУУ “КПІ”
2016

Методичні вказівки виконання практичних робіт з дисципліни «Обчислювальна техніка та програмування. Частина 2» для студентів спеціальності 141 Електроенергетика, електротехніка та електромеханіка / Уклад.: Д.В. Настенко, А.Б. Нестерко, Г.О. Труніна – Київ: НТУУ “КПІ”, 2016.

Гриф надано Вченою радою ФЕА НТУУ “КПІ”
(Протокол № 10 від 30 червня 2016 р.)

Навчальне електронне видання

МЕТОДИЧНІ ВКАЗІВКИ ТА ЗАВДАННЯ
до виконання практичних робіт з дисципліни

«ОБЧИСЛЮВАЛЬНА ТЕХНІКА ТА ПРОГРАМУВАННЯ. ЧАСТИНА 2»
141 Електроенергетика, електротехніка та електромеханіка

Укладачі: Настенко Дмитро Васильович, ст. викл.,
Нестерко Артем Борисович, к.т.н., ст. викл.
Труніна Ганна Олексіївна, асистент.

Відповідальний
Редактор О.С. Яндульський, професор, д.т.н.

Рецензент Т.Л. Кацадзе, канд. техн. наук

За редакцією укладачів

Зміст

Вступ

Microsoft .NET — програмна технологія компанії Microsoft, яка є платформою для створення як звичайних додатків, так і веб- додатків. .NET-додатки можуть розроблятися і виконуватися як в середовищі операційних систем Microsoft, так і в інших включаючи Mac OS X, різні дистрибутиви Linux, Solaris, а також на мобільних пристроях iOS і Android (через API-інтерфейс MonoTouch).

C# – це об'єктно-орієнтована мова програмування, розроблена в 1998-2001 роках групою інженерів під керівництвом Андерса Хейлсберга в компанії Microsoft як мова розробки додатків для платформи Microsoft .NET. Синтаксис C# близький до C++ і Java. На сьогодні C# є флагманською мовою корпорації Microsoft для програмування в середовищі Windows.

Основним інтегрованим середовищем розробки, тобто комп'ютерною програмою, що допомагає програмістові розробляти нове програмне забезпечення, для C# є Visual Studio. Альтернативними середовищами програмування є SharpDevelop та MonoDevelop.

C # – не єдина мова, яка може використовуватися для побудови .NET-додатків. Середовище Visual Studio дає змогу використовувати п'ять мов, а саме – C #, Visual Basic, C ++/CLI, JavaScript і F#. Існують .NET-компілятори які призначені для мов Ruby, Python, Pascal тощо.

Visual Studio працює на платформі Windows і орієнтована на створення Windows- і веб-додатків. У другій частині курсу ми будемо вивчати принципи обєктно-орієнтованого програмування та створювати Windows-додатки.

Практичне заняття №1. Методи. Параметри методів.

Метод - це функціональний елемент класу, який реалізує обчислення або інші дії, що виконуються класом чи примірником. Методи визначають поведінку класу.

Метод являє собою закінчений фрагмент коду, до якого можна звернутися по імені. Він описується один раз, а викликатися може стільки разів, скільки необхідно. Один і той же метод може обробляти різні дані, передані йому в якості аргументів.

Синтаксис методу:

[Атрибути] [специфікатори] тип імяметода ([параметри])

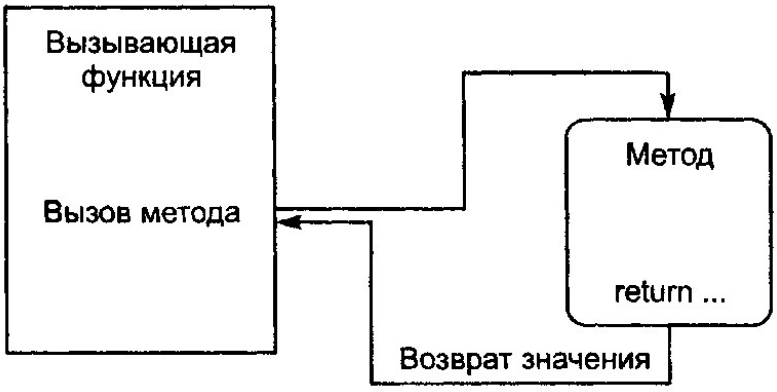
тело_метода

Розглянемо основні елементи опису методу. Перший рядок являє собою заголовок методу. Тіло методу, що задає дії, що виконуються методом, найчастіше являє собою блок - послідовність операторів у фігурних скобках1.

При описі методів можна використовувати специфікатори 1-7 з табл. 5.2, що мають той же зміст, що й для полів, а також специфікатори virtual, sealed, override, abstract і extern, які будуть розглянуті в міру необхідності. Найчастіше для методів задається специфікатор доступу public, адже методи складають інтерфейс класу - те, з чим працює користувач, тому вони повинні бути доступні.

```
public double Gety()          // метод для получения поля y из листинга 5.1
   {
       return y;
   }
```

Тип визначає, значення якого типу обчислюється за допомогою методу. Часто вживається термін «метод повертає значення», оскільки після виконання методу відбувається повернення в те місце викликає функції, звідки був викликаний метод, і передача туди значення виразу, записаного в операторі return. Якщо метод не повертає ніякого значення, в його заголовку задається тип void, а оператор return відсутня.



Виклик методу

Параметри використовуються для обміну інформацією з методом. Параметр являє собою локальну змінну, яка при виклику методу приймає значення відповідного аргументу. Область дії параметра - весь метод.

Наприклад, щоб обчислити значення синуса для речовій величини x, ми передаємо її в якості аргументу в метод Sin класу Math, а щоб вивести значення цієї змінної на екран, ми передаємо її в метод WriteLine класу Console:

```
double x = 0.1;
double y = Math.Sin(x);
Console.WriteLine(x);
```

При цьому метод Sin повертає в точку свого виклику речовинне значення синуса, яке присвоюється змінної y, а метод WriteLine нічого не повертає.

Параметри, описувані в заголовку методу, визначають безліч значень аргументів, які можна передавати в метод. Список аргументів при виклику ніби накладається на список параметрів, тому вони повинні попарно відповідати один одному. Правила відповідності докладно розглядаються в наступних розділах.

Для кожного параметра повинні задаватися його тип та ім'я. Наприклад, заголовок методу Sin виглядає наступним чином:

```
public static double Sin( double a );
```

Ім'я методу укупі з кількістю, типами і специфікаторами його параметрів являє собою сигнатуру методу - те, по чому один метод відрізняють від інших. У класі не повинно бути методів з однаковими сигнатурами.

У лістингу 5.2 в клас Demo додані методи установки і отримання значення поля y (насправді для подібних цілей використовуються не методи, а властивості, які розглядаються трохи пізніше). Крім того, статичне поле s закрито, тобто визначене за умовчанням як private, а для його отримання описаний метод Gets, являло собою приклад статичного методу.

Листинг 5.2. Простейшие методы

```
using System;
namespace ConsoleApplication1
{
    class Demo
    {
        public int a = 1;
        public const double c = 1.66;
        static string s = "Demo";
        double y;

        public double Gety()                // метод получения поля y
        {
            return y;
        }

        public void Sety( double y_ )       // метод установки поля y
        {
            y = y_;
        }

        public static string Gets()         // метод получения поля s
        {
            return s;
        }
    }

    class Class1
    {   static void Main()
        {
            Demo x = new Demo();
            x.Sety(0.12);                   // вызов метода установки поля y


           Console.WriteLine( x.Gety() );    // вызов метода получения поля y
           Console.WriteLine( Demo.Gets() ); // вызов метода получения поля s
//         Console.WriteLine( Gets() );      // при вызове из др. метода этого объекта
       }
    }
}
```

Як бачите, методи класу мають безпосередній доступ до його закритим полях. Метод, описаний зі специфікатором static, повинен звертатися тільки

до статичних полів класу. Зверніть увагу на те, що статичний метод викликається через ім'я класу, а звичайний - через ім'я екземпляра.

Розглянемо більш детально, яким чином метод обмінюється інформацією з викликав його кодом. При виклику методу виконуються наступні дії:

1. Обчислюються виразів, що стоять на місці аргументів.

2. Виділяється пам'ять під параметри методу відповідно до їх типом.

3. Кожному з параметрів зіставляється відповідний аргумент (аргументи як би накладаються на параметри і заміщають їх).

4. Виконується тіло методу.

5. Якщо метод повертає значення, воно передається в точку виклику; якщо метод має тип voi d, управління передається на оператор, наступний після виклику.

При цьому перевіряється відповідність типів аргументів і параметрів і при необхідності виконується їх перетворення. При невідповідності типів видається діагностичне повідомлення. Лістинг ілюструє цей процес.

```
using System;
namespace ConsoleApplication1
{   class Class1
    {
        static int Max(int a, int b)  // метод выбора максимального значения
        {
            if ( a > b ) return a;
            else         return b;
        }
        static void Main()
        {
            int a = 2, b = 4;
            int x = Max( a, b );                 // вызов метода Max
            Console.WriteLine( x );              // результат: 4

            short t1 = 3, t2 = 4;
            int y = Max( t1, t2 );              // вызов метода Max
            Console.WriteLine( y );             // результат: 4

            int z = Max( a + t1, t1 / 2 * b );  // вызов метода Max
            Console.WriteLine( z );             // результат: 5
        }
    }
}
```

У класі описаний метод Max, який вибирає найбільшу з двох переданих йому значень. Параметри описані як a і Ь. У методі Main виконуються три

виклики Max. В результаті першого виклику методу Max передаються два аргументи того ж типу, що й параметри, у другому виклику - аргументи сумісного типу, в третьому - вираження.

Кількість аргументів повинна відповідати кількості параметрів. На імена ніяких обмежень не накладається: імена аргументів можуть як збігатися, так і не збігатися з іменами параметрів.

Існують два способи передачі параметрів: за значенням і за посиланням.

При передачі за значенням метод отримує копії значень аргументів, і оператори методу працюють з цими копіями. Доступу до початкових значень аргументів у методу немає, а отже, немає і можливості їх змінити.

При передачі по посиланню (за адресою) метод отримує копії адрес аргументів, він здійснює доступ до комірок пам'яті за цими адресами і може змінювати вихідні значення аргументів, модифікуючи параметри.

У C # для обміну даними між викликає і викликається функціями передбачено чотири типи параметрів:

- параметри-значення;
- параметри-посилання - описуються за допомогою ключового слова ref;
- вихідні параметри - описуються за допомогою ключового слова out;
- параметри-масиви - описуються за допомогою ключового слова params.

Ключове слово передує опису типу параметра. Якщо воно опущено, параметр вважається параметром-значенням. Параметр-масив може бути тільки один і повинен розташовуватися останнім у списку, наприклад:

```
public int Calculate( int a, ref int b, out int c, params int[] d ) …
```

Про параметри-масивах ми будемо говорити пізніше, в главі 7 (див. С. 154), а зараз розглянемо інші типи параметрів.

Параметри-значення

Параметр-значення описується в заголовку методу наступним чином: тип ім'я

Приклад заголовка методу, що має один параметр-значення цілого типу: void P (int x)

Ім'я параметра може бути довільним. Параметр x являє собою локальну змінну, яка отримує своє значення з викликає функції при виклику методу. У метод передається копія значення аргументу.

Механізм передачі наступний: з комірки пам'яті, в якій зберігається змінна, передана в метод, береться її значення і копіюється в спеціальну область пам'яті - область параметрів. Метод працює з цією копією, отже, доступу до комірки, де зберігається сама змінна, не має. По завершенні роботи методу область параметрів звільняється. Таким чином, для параметрів-значень використовується, як ви здогадалися, передача за значенням. Ясно, що цей спосіб годиться тільки для величин, які не повинні змінитися після виконання методу, тобто для його вихідних даних.

При виклику методу на місці параметра, переданого за значенням, може перебувати вираз, а також, звичайно, його окремі випадки - змінна або константа. Має існувати неявне перетворення типу виразу до типу параметра1.

Наприклад, нехай в викликає функції описані змінні і їм до виклику методу привласнені значення:

```
int    x = 1;
sbyte  c = 1;
ushort y = 1;
```

Тоді наступні виклики методу P, заголовок якого був описаний раніше, будуть синтаксично правильними:

```
P( x );    P( c ),    P( y );    P( 200 );    P( x / 4 + 1 );
```

Параметри-посилання

У багатьох методах всі величини, які метод повинен отримати в якості вихідних даних, описуються в списку параметрів, а величина, яку обчислює метод як результат своєї роботи, повертається в викликає код за допомогою оператора return. Очевидно, що якщо метод повинен повертати більше однієї величини, такий спосіб не годиться. Ще одна проблема виникає, якщо в методі потрібно змінити значення будь-яких переданих до нього величин. У цих випадках використовуються параметри-посилання.

Ознакою параметра-посилання є ключове слово ref перед описом параметра:

ref тип ім'я

Приклад заголовка методу, що має один параметр-посилання цілого типу: void P (ref int x)

При виклику методу в область параметрів копіюється НЕ значення аргументу, а його адресу, і метод через нього має доступ до осередку, у якій зберігається аргумент. Таким чином, параметри-посилання передаються за адресою (частіше вживається термін «передача за посиланням»). Метод працює безпосередньо з змінної з викликає функції і, отже, може її змінити, тому якщо в методі потрібно змінити значення параметрів, вони повинні передаватися тільки по посиланню.

При виклику методу на місці параметра-посилання може знаходитися тільки посилання на ініціалізувала змінну точно того ж типу. Перед ім'ям параметра вказується ключове слово ref.

Вихідні дані передавати в метод за посиланням не рекомендується, щоб виключити можливість їх ненавмисного зміни.

Проілюструємо передачу параметрів-значень і параметрів-посилань на прикладі (лістинг 5.4).

```
using System;
namespace ConsoleApplication1
{   class Class1
    {
        static void P( int a, ref int b )
        {
            a = 44; b = 33;
            Console.WriteLine( "внутри метода {0} {1}", a, b );
        }
        static void Main()
        {
            int a = 2, b = 4;
            Console.WriteLine( "до вызова     {0} {1}", a, b );
            P( a, ref b );
            Console.WriteLine( "после вызова  {0} {1}", a, b );
        }
```

Як бачите, значення змінної a в функції Main не змінилося, оскільки змінна передавалася за значенням, а значення змінної b змінилося тому, що вона була передана за посиланням.

Дещо інша картина вийде, якщо передавати в метод не величини значущих типів, а екземпляри класів, тобто величини посилальних типів. Як ви пам'ятаєте, змінна-об'єкт насправді зберігає посилання на дані, розташовані в динамічної пам'яті, і саме ця посилання передається в метод або за адресою, або за значенням. В обох випадках метод отримує в своє розпорядження фактична адреса даних і, отже, може їх змінити.

Різниця між передачею об'єктів за значенням і за посиланням полягає в тому, що в останньому випадку можна змінити саму посилання, тобто після виклику методу вона може вказувати на інший об'єкт.

Вихідні параметри

Досить часто виникає необхідність в методах, які формують кілька величин, наприклад, якщо в методі створюються об'єкти або инициализируются ресурси. У цьому випадку стає незручним обмеження параметров- посилань: необхідність присвоювання значення аргументу до виклику методу. Це обмеження знімає специфікатор out. Параметри, що мають цей специфікатор, має бути обов'язково присвоєно значення всередині методу, компілятор за цим стежить. Зате в зухвалій коді можна обмежитися описом змінної без ініціалізації.

Змінимо опис другого параметра в лістингу 5.4 так, щоб він став вихідним.

```
using System;
namespace ConsoleApplication1
{   class Class1
    {
        static void P( int a, out int b )
        {
            a = 44; b = 33;
            Console.WriteLine( "внутри метода {0} {1}", a, b );
        }
        static void Main()
        {
```

```
            int a = 2, b;
            P( a, out b );
            Console.WriteLine( "после вызова  {0} {1}", a, b );
        }
    }
}
```

При виклику методу перед відповідним параметром теж вказується ключове слово out.

Практичне заняття №2. Конструктори, деструктори та властивості.

У наведених вище прикладах програм змінні екземпляра кожного об'єкта типу доводилося задавати вручну, використовуючи, зокрема, наступну послідовність операторів.

```
house.Occupants = 4;
house.Area = 2500;
house.Floors = 2;
```

Такий прийом зазвичай не застосовується в професійно написаному коді C#. Крім того, він супроводжується помилками, коли можна просто забути ініціалізувати одне з полів. Існує кращий спосіб вирішити подібну задачу: скористатися конструктором.

Конструктор призначений для ініціалізації об'єкту. Він викликається автоматично при створенні об'єкта класу за допомогою операції new. Ім'я конструктора обов’язково збігається з ім'ям класу.

Нижче наведена загальна форма конструктора:

```
[специфікатори] ім'я_класу ([список_параметрів])
тіло_конструктора
```

Нижче перераховані властивості конструкторів:

- конструктор не повертає значення, навіть типу void;
- клас може мати кілька конструкторів з різними параметрами для різних видів ініціалізації;
- якщо програміст не вказав жодного конструктора то застосовується конструктор за замовчуванням, у якому полям значущих типів присвоюється нуль, а полям посилальних типів - значення null.

Як правило, конструктор використовується для завдання початкових значень змінних екземпляра, визначених в класі, або ж для виконання будь-

яких інших процедур, які потрібні для створення повністю сформованого коректного об'єкта. Для конструктора зазвичай задають специфікатор доступу public, оскільки конструктори, в основному, викликаються з інших класів. Список параметрів конструктора може бути порожнім, або містити один чи більше параметрів.

У всіх класів є конструктори, незалежно від того, визначте ви їх чи ні, оскільки в C# автоматично надається конструктор, який використовується за замовчуванням і у якому всі змінні екземпляра ініціалізуються їх значеннями за замовчуванням. Для більшості типів даних значенням за замовчуванням є нульовим, для типу bool – значення false, а для типів-посилань – значення null. Як тільки ви визначите свій власний конструктор, то конструктор за замовчуванням більше не викликається.

Дотепер ми задавали початкові значення полів класу при описі класу. Це зручно в тому випадку, коли для всіх екземплярів класу початкові значення деякого поля однакові. Якщо ж при створенні об'єктів потрібно присвоювати полю різні значення, це слід робити в конструкторі. У лістингу в клас Demo доданий конструктор, а поля зроблені закритими (непотрібні в даний момент елементи опущені). У програмі створюються два об'єкти з різними значеннями полів.

```
using System;
namespace ConsoleApplication1
{
    class Demo
    {
        public Demo( int a, double y )          // конструктор с параметрами
        {
            this.a = a;
            this.y = y;
        }

        public double Gety()                    // метод получения поля y
        {
            return y;
        }

        int a;
        double y;
    }

    class Class1
    {   static void Main()
        {
            Demo a = new Demo( 300, 0.002 );     // вызов конструктора
            Console.WriteLine( a.Gety() );       // результат: 0.002
            Demo b = new Demo( 1, 5.71 );        // вызов конструктора
            Console.WriteLine( b.Gety() );       // результат: 5.71
        }
```

Часто буває зручно задати в класі кілька конструкторів, щоб забезпечити можливість ініціалізації об'єктів різними способами. У цьому випадку використовуються параметризовані конструктори. У конструктор параметри вводяться таким же чином, як і в метод. Для цього достатньо оголосити їх в дужках після імені конструктора.

Нижче наведено приклад застосування параметризованих конструктора MyClass.

```
// Параметризованих конструктор. using System;
class MyClass
{
  public int x;
        public MyClass (int i) {
                x = i;
        }
}
class ParmConsDemo
{
  static void Main ()
  {
```

```
        MyClass t1 = new MyClass (10);
        MyClass t2 = new MyClass (88);
        Console.WriteLine (t1.x + "" + t2.x);
    }
}
```

При виконанні цього коду виходить наступний результат.

10 88

В даному варіанті конструктора MyClass () визначено параметр і, за допомогою якого инициализируется змінна екземпляра x. Тому при виконанні наступного рядка коду:

MyClass t1 = new MyClass (10);

параметру і передається значення, яке потім присвоюється змінної x.

Всі конструктори повинні мати різні сигнатури.

Тепер, коли ви ближче ознайомилися з класами і їх конструкторами, повернемося до оператора new, щоб розглянути його більш детально. Відносно класів загальна форма оператора new така:

new ім'я_класу (список_аргументів)

де ім'я_класу позначає ім'я класу, що реалізується у вигляді примірника його об'єкта. А ім'я_класу з подальшими дужками позначає конструктор цього класу. Якщо в класі не визначений його власний конструктор, то в операторі new буде використаний конструктор, що надається в C # за умовчанням. Отже, оператор new може бути використаний для створення об'єкта, що відноситься до класу будь-якого типу.

Оперативна пам'ять не нескінченна, і тому цілком можливо, що оператору new не вдасться розподілити пам'ять для об'єкта через брак наявної оперативної пам'яті. В цьому випадку виникає виняткова ситуація під час виконання. У прикладах програм, наведених у цій книзі, ситуація, пов'язана з вичерпанням оперативної пам'яті, не враховується, але при написанні реальних програм таку можливість, ймовірно, доведеться брати до уваги.

Якщо один з конструкторів виконує будь-які дії, а інший повинен робити те ж саме плюс ще що-небудь, зручно викликати один конструктор з

іншого. Для цього використовується вже відоме вам ключове слово this в іншому контексті, наприклад:

```
class Demo
{
    public Demo( int a )                          // конструктор 1
    {
        this.a = a;
    }
    public Demo( int a, double y ) : this( a )    // вызов конструктора 1
    {
        this.y = y;
    }
    ...
}
```

Конструкція, що знаходиться після двокрапки, називається Ініціалізатором, тобто тим кодом, який виповнюється до початку виконання тіла конструктора.

Як ви пам'ятаєте, всі класи в C # мають спільного предка - клас object. Конструктор будь-якого класу, якщо не вказано ініціалізатор, автоматично викликає конструктор свого предка. Це можна зробити і явним чином за допомогою ключового слова base, що позначає конструктор базового класу. Таким чином, перший конструктор з попереднього прикладу можна записати і так:

```
public Demo( int a ) : base()              // конструктор 1
{
    this.a = a;
}
```

До цих пір мова йшла про «звичайних» конструкторів, або конструкторів екземпляра. Існує інший тип конструкторів - статичні конструктори, або конструктори класу. Конструктор екземпляра ініціалізує дані екземпляра, конструктор класу - дані класу.

Статичний конструктор не має параметрів, його не можна викликати явним чином. Система сама визначає момент, в який потрібно його виконати. Гарантується тільки, що це відбувається до створення першого примірника об'єкта і до виклику будь-якого статичного методу.

Деякі класи містять тільки статичні дані, і, отже, створювати екземпляри таких об'єктів не має сенсу. Щоб підкреслити цей факт, в першій версії C #

описували порожній закритий (private) конструктор. Це запобігало спроби створення екземплярів класу. У лістингу 5.7 наведено приклад класу, який служить для групування величин. Створювати екземпляри цього класу заборонено.

```
using System;
namespace ConsoleApplication1
{
    class D
    {
        private D(){}          // закрытый конструктор
        static D()             // статический конструктор
        {
            a = 200;
        }
        static int a;
        static double b = 0.002;
        public static void Print()
        {
            Console.WriteLine( "a = " + a );
            Console.WriteLine( "b = " + b );
        }
        ...
    }

    class Class2
    {   static void Main()
        {
            D.Print();
//          D d = new D();          // ошибка: создать экземпляр невозможно
        }
    }
}
```

У версію 2.0 введена можливість описувати статичний клас, тобто клас з модифікатором static. Примірники такого класу створювати заборонено, і крім того, від нього заборонено успадковувати. Всі елементи такого класу повинні явним чином оголошуватися з модифікатором static (константи і вкладені типи класифікуються як статичні елементи автоматично).Конструктор примірника для статичного класу здавати, природно, забороняється.

Властивості. Взаємодія з автоматичними властивостями.

Властивість є елементом класу, що поєднує в собі поле з методами доступу до нього. Властивості служать для організації доступу до полів класу. Як правило, властивість пов'язана із закритим полем класу і визначає методи його отримання та установки.

Властивість складається з імені і так званих аксесорів get і set. Аксесори служать для зчитування і установки значення змінної. Головна перевага властивості над спеціальними методами полягає в тому, що її ім'я може бути використано в виразах і операторах присвоювання аналогічно імені звичайної змінної. При зверненні до властивості по імені автоматично викликаються її аксесори get і set.

Нижче наведена загальна форма властивості:

```
[ атрибути ] [ специфікатори ] тип ім’я_властивості
{
[ специфікатор ] [ get код_доступу ]
[ специфікатор ] [ set код_доступу ]
}
```

де тип позначає конкретний тип властивості, наприклад int.

Значення специфікаторів для властивостей і методів аналогічні. Найчастіше властивості оголошуються як відкриті (зі специфікатором public), оскільки вони реалізують взаємодію зовнішнього коду з даними класу.

Код доступу є блоком операторів, які виконуються при отриманні (get) або установці (set) властивості.

Як тільки властивість визначено, будь-яке звернення до властивості по імені приведе до автоматичного виклику відповідного аксесора. Крім того, аксесор set має неявний параметр value, який містить значення, що присвоюється властивості.

Властивості не визначають місце в пам'яті для зберігання полів, а лише управляють доступом до полів. Сама властивість не описує поле, і тому поле повинно бути визначено незалежно від властивості (винятком з цього правила є автоматичні властивості).

Нижче наведено простий приклад програми, в якій визначається властивість Power, призначена для доступу до поля _power. В даному прикладі властивість використовується для обмеження діапазону допустимих значень потужності лампи.

```
using System;

namespace ClassElementTest
{
   class Lamp
   {
      private double _power;
      public double NominalPower;

      // Властивість для доступу до поля _power
      public double Power
      {
         get { return _power; }
         set {
            if (value>=0 && value<=NominalPower)
            {
               _power = value;
            }
         }
      }

      // Властивість без базового поля. Призначена для розрахунку значення струму
      public double Current
      {
         get { return _power/220; } // Приймемо, що I=P/U, U=220 В
      }
   }
   class Program
   {
      static void Main(string[] args)
      {
         var myLamp = new Lamp();
         myLamp.NominalPower = 100;
         // Задаєм значення за допомогою властивості
         myLamp.Power = 83.5;
         Console.WriteLine("Потужність лампи: {0} Вт, струм лампи: {1:n3} А",
            myLamp.Power, myLamp.Current);
      }
   }
}
```

Розглянемо наведений вище код більш докладно. У цьому коді визначається закрите поле _power і властивість Power, котра управляє доступом до поля _power.

Властивість Power вказано як public, а отже, вона доступна з коду за межами класу. Аксесор get цієї властивості просто повертає значення поля _power, тоді як аксесор set встановлює значення в поле _power тільки в тому випадку, якщо це значення відповідає обмеженням. Таким чином, властивість Power контролює значення, які можуть зберігатися в полі _power. У цьому, власне, і полягає основне призначення властивостей.

Починаючи з версії C# 3.0, з'явилася можливість створювати прості властивості, не вдаючись до явного визначення поля, яким управляє властивість. У цьому випадку базове поле для властивості автоматично надає компілятор. Така властивість називається автоматичною і має таку загальну форму:

```
[ атрибути ] [ специфікатори ] тип ім’я_властивості
{get; set; }
```

Зверніть увагу на те, що після позначень аксесорів get і set відразу ж ставиться крапка з комою, а тіло у них відсутнє. Такий синтаксис вказує компілятору створити автоматично поле для зберігання значення властивості. Таке поле недоступне безпосередньо і не має імені.

Нижче наведено приклад опису автоматичної властивості:

```
public double Voltage { get; set; }
```

Як бачите, в цьому рядку змінна явно не оголошується. Компілятор автоматично створює анонімне поле, в якому зберігається значення властивості.

На відміну від звичайних властивостей, автоматична властивість не може бути доступною тільки для читання або тільки для запису. При оголошенні цієї властивості необхідно вказувати обидва аксесори – get і set. Зробити автоматичну властивість доступною тільки для читання або тільки для запису можна оголосивши непотрібний аксесор як private.

Незважаючи на очевидні зручності автоматичних властивостей, їх застосування обмежується в основному тими ситуаціями, в яких не потрібно керувати установкою або отриманням значень з полів. Базове поле автоматичної властивості недоступне безпосередньо. Це означає, що на значення, яке може мати автоматична властивість, не можна накласти ніяких обмежень.

Властивостями притаманний ряд обмежень. По-перше, властивість не визначає місце для зберігання даних, і тому не може бути передана методу в якості параметра ref або out. По-друге, властивість не повинна змінювати стан базової змінної при виклику аксесора get. І хоча це правило не перевіряється компілятором, його порушення вважається семантичної помилкою.

За замовчуванням доступність аксесорів set і get є такою ж, як і у властивості, частиною якої вони є. Так, якщо властивість оголошується як public, то за замовчуванням її аксесори set і get також стають відкритими (public). Проте для аксесорів set або get можна вказати власний специфікатор доступу, наприклад private. Доступність аксесорів, що визначається таким специфікатором, повинна бути більш обмеженою, ніж доступність властивості.

Існує ряд причин, за яких потрібно обмежити доступність аксесорів. Припустимо, що потрібно надати вільний доступ до значення властивості, але разом з тим дати можливість змінювати цю властивість тільки елементам її класу. Для цього достатньо оголосити аксесор set даної властивості як private.

Практичне заняття №3. Колекції та списки.

У бібліотеках більшості сучасних об'єктно-орієнтованих мов програмування представлені стандартні класи, що реалізують основні абстрактні структури даних. Такі класи називаються колекціями, або контейнерами. Для кожного типу колекції визначені методи роботи з її елементами, які не залежать від конкретного типу даних, які зберігаються в колекції, тому один і той же вид колекції можна використовувати для зберігання даних різних типів. Використання колекцій дозволяє скоротити терміни розробки програм і підвищити їх надійність.

Кожен вид колекції підтримує свій набір операцій над даними, і швидкодія цих операцій може бути різним. Вибір виду колекції залежить від того, що потрібно робити з даними в програмі і які вимоги пред'являються до її швидкодії. Наприклад, при необхідності часто вставляти і видаляти елементи з середини послідовності слід використовувати список, а не масив, а якщо включення елементів виконується головним чином в кінець або початок послідовності - черга. Тому вивчення можливостей стандартних колекцій і їх грамотне застосування є необхідними умовами створення ефективних і професійних програм.

У бібліотеці .NET визначені стандартні класи, що реалізують більшість перерахованих раніше абстрактних структур даних. Основні простори імен, в яких описані ці класи, – **System.Collections, System.Collections.Specialized** і **System.Collections.Generic**.

Всі колекції розроблені на основі набору чітко визначених інтерфейсів. Деякі вбудовані реалізації таких інтерфейсів, в тому числі **ArrayList, Hashtable, Stack** і **Queue**, можуть застосовуватися в початковому вигляді і без будь-яких змін. Є також можливість реалізувати власну колекцію, хоча потреба в цьому виникає вкрай рідко.

У середовищі .NET Framework підтримуються п'ять типів колекцій:

- неузагальнені,
- спеціальні,

- з поразрядною організацією,
- узагальнені,
- паралельні.

Неузагальнені колекції реалізують ряд основних структур даних, включаючи динамічний масив, стек, чергу, а також словники, в яких можна зберігати пари "ключ-значення". Відносно неузагальнених колекцій важливо мати на увазі наступне: вони оперують даними типу **object**.

Таким чином, неузагальнені колекції можуть служити для зберігання даних будь-якого типу, причому в одній колекції допускається наявність різнотипних даних. Очевидно, що такі колекції не типізовані, оскільки в них зберігаються посилання на дані типу **object**. Класи і інтерфейси неузагальнених колекцій знаходяться в просторі імен **System.Collections**.

Спеціальні колекції оперують даними конкретного типу або ж роблять це якимось особливим чином. Наприклад, є спеціальні колекції для символьних рядків, а також спеціальні колекції, в яких використовується односпрямований список. Спеціальні колекції оголошуються в просторі імен **System.Collections.Specialized**.

У прикладному інтерфейсі **Collections API** визначена одна колекція з поразрядною організацією - це **BitArray**. Колекція типу **BitArray** підтримує порозрядні операції, тобто операції над окремими двійковими розрядами, наприклад, І йди виключає АБО, а отже, вона істотно відрізняється своїми можливостями від інших типів колекцій. Колекція типу **BitArray** оголошується в просторі імен **System.Collections**.

Узагальнені колекції забезпечують узагальнену реалізацію декількох стандартних структур даних, включаючи списки, стеки, черги і словники. Такі колекції є типізованими в силу їх узагальненого характеру. Це означає, що в узагальненій колекції можуть зберігатися тільки такі елементи даних, які сумісні за типом з даної колекцією. Завдяки цьому виключається випадкова розбіжність типів. Узагальнені колекції оголошуються в просторі імен **System.Collections.Generic**.

Як правило, класи узагальнених колекцій є узагальненими еквівалентами класів неузагальнених колекцій, хоча це відповідність не є взаємно однозначною. Наприклад, в класі узагальненої колекції **LinkedList** реалізується двонаправлений список, тоді як в неузагальнених еквіваленті його не існує. У деяких випадках одні й ті ж функції існують паралельно в класах узагальнених і неузагальнених колекцій, хоча і під різними іменами. Так, узагальнений варіант класу **ArrayList** називається **List**, а узагальнений варіант класу **HashTable** – **Dictionary**.

У табл. 5.1 перераховані основні колекції, описані в просторі **System.Collections.Generic**.

Таблиця 5.1 – Колекції простору імен System.Collections.Generic

Клас	**Опис**
Dictionary <Tkey, TValue>	Зберігає пари "ключ-значення". Забезпечує такі ж функціональні можливості, як і неузагальнений клас Hashtable
HashSet <T>	Зберігає ряд унікальних значень, використовуючи хеш таблицю
LinkedList <T>	Зберігає елементи в двонаправленому списку
List <T>	Створює динамічний масив. Забезпечує такі ж функціональні можливості, як і неузагальнений клас ArrayList
Queue <T>	Створює чергу. Забезпечує такі ж функціональні можливості, як і неузагальнений клас Queue
SortedDictionary <TKey, TValue>	Створює відсортований список з пар "ключ-значення"
SortedList <TKey, TValue>	Створює відсортований список з пар "ключ-значення". Забезпечує такі ж функціональні можливості, як і неузагальнений клас SortedList
SortedSet <T>	Створює відсортовану множину

Stack <T>	Створює стек. Забезпечує такі ж функціональні можливості, як і неузагальнених клас Stack

У колекції можна зберігати не тільки об'єкти вбудованих типів. В них допускається зберігати об'єкти будь-якого типу, включаючи об'єкти класів, що визначаються користувачем.

Завдяки тому, що всі типи успадковують від класу **object**, в неузагальнених колекції можна зберігати об'єкти будь-якого типу. Для того щоб зберегти об'єкти класів, що визначаються користувачем, в типізованій колекції, доведеться скористатися класами узагальнених колекцій.

У колекції, для якої оголошено тип елементів, завдяки поліморфізму можна зберігати елементи будь-якого похідного класу, але не елементи інших типів.

Здавалося б, у порівнянні зі звичайними колекціями це обмеження, а не універсальність, однак на практиці колекції, в яких дійсно потрібно зберігати значення різних, не пов'язаних межу собою типів, майже не використовуються. Перевагою ж такого обмеження є те, що компілятор може виконати контроль типів під час компіляції, а не виконання програми, що підвищує її надійність і спрощує пошук помилок.

Використання стандартних узагальнених колекцій для зберігання і обробки даних є хорошим стилем програмування, оскільки дозволяє скоротити терміни розробки програм і підвищити їх надійність. Рекомендується ретельно вивчити властивості і методи цих класів і вибирати найбільш підходящі в залежності від розв'язуваної задачі.

Розглянемо детально клас **List<T>**. У класі **List<T>** реалізується узагальнений динамічний масив.

В класі **List<T>** визначено наступні властивості:

```
int Count {get; }
bool IsReadOnly {get; }
```

Властивість **Count** містить ряд елементів, що зберігаються в даний момент в колекції. А властивість **IsReadOnly** має логічне значення **true**, якщо колекція доступна тільки для читання. Якщо ж колекція доступна як для читання, так і для запису, то ця властивість має логічне значення **false**.

У класі **List<T>** визначається також властивість **Capacity**. Це властивість оголошується наступним чином.

```
public int Capacity {get; set; }
```

Властивість **Capacity** дозволяє встановити і отримати ємність колекції в якості динамічного масиву. Ця ємність дорівнює кількості елементів, які може містити колекція до її вимушеного розширення. Така колекція розширюється автоматично, і тому встановлювати її ємність вручну необов'язково. Але з міркувань ефективності це іноді можна зробити, якщо заздалегідь відомо кількість елементів колекції. Завдяки цьому виключаються витрати на виділення додаткової пам'яті.

У класу **List<T>** є такі конструктори.

```
public List ()
public List (IEnumerable <T> collection)
public List (int capacity)
```

Перший конструктор створює порожню колекцію класу клас List <T>з обраною за замовчуванням ємністю. Другий конструктор створює колекцію типу **List** з кількістю елементів, яка визначається розміром параметра **collection**. Третій конструктор створює колекцію типу **List**, що має початкову ємність, яка задається параметром **capacity**.

Ємність колекції, яка створюється у вигляді динамічного масиву, може збільшуватися автоматично при додаванні в неї елементів.

У класі **List<T>** визначається ряд методів, найбільш важливі з яких перераховані в табл. 5.2.

Методи, визначені в класі List <T>

Метод	Опис
void Add (T item)	Додає елемент item в колекцію. Генерує виняток NotSupportedException, якщо колекція доступна тільки для читання
void Clear ()	Видаляє всі елементи з колекції
bool Contains (T item)	Повертає логічне значення true, якщо колекція містить елемент item, а інакше - логічне значення false
void CopyTo (T [] array, int arraylndex)	Копіює вміст колекції в масив array, починаючи з елемента, що вказується за індексом arraylndex
bool Remove (T item)	Видаляє перше входження елемента item в колекції. Повертає логічне значення true, якщо елемент item видалений. А якщо цей елемент не знайдено в колекції, то повертається логічне значення false
int IndexOf (T item)	Повертає індекс першого входження елемента item в колекції. Якщо елемент item не виявлений, то метод повертає значення -1
void Insert (int index, T item)	Вставляє в колекцію елемент item за індексом index
void RemoveAt (int index)	Видаляє з колекції елемент, розташований за індексом index
public virtual void AddRange (ICollection collection)	Додає елементи з колекції collection типу ArrayList
public virtual int BinarySearch (T item)	Виконує пошук в колекції значення, що задається параметром item. Повертає індекс знайденого

	елемента. Якщо шукане значення не знайдено, повертається від'ємне значення. Список повинен бути відсортований
public List <T> GetRange (int Index, int count)	Повертає частину колекції. Частина колекції починається з елемента за індексом index, і включає кількість елементів, що задається параметром count
public int IndexOf (T item)	Повертає індекс першого входження елемента item в колекції. Якщо шуканий елемент не виявлений, повертається значення -1
public void InsertRange (int index, IEnumerable <T> collection)	Вставляє елементи колекції collection починаючи з елемента за індексом index
public item int LastIndexOf (T)	Повертає індекс останнього входження елемента item в колекції. Якщо шуканий елемент не виявлений, повертається значення -1
public void RemoveRange (int index, int count)	Видаляє частину колекції, починаючи з елемента за індексом index, і включаючи кількість елементів, яка визначається параметром count
public void Reverse ()	Повертає колекцію в зворотному порядку
public void Reverse (int index, int count)	Повертає колекцію в зворотному порядку, починаючи з елемента за індексом index, і включаючи кількість елементів, яке визначається параметром count
public void Sort ()	Сортує колекцію за зростанням
public void Sort (IComparer <T> comparer)	Сортує колекцію, використовуючи для порівняння спосіб, що задається параметром comparer. Якщо параметр comparer має пусте значення, то для порівняння використовується

	спосіб, який обирається за замовчуванням
public T [] ToArray()	Повертає масив, який містить копії елементів колекції

Для циклічного звернення до елементів колекції, що представляє собою групу об'єктів, служить оператор **foreach**. Нижче наведена загальна форма оператора циклу **foreach**.

```
foreach (тип імя_змінної_циклу in ім’я_колекції) оператор;
```

Тут тип *імя_змінної_циклу* позначає тип і ім'я змінної управління циклом, яка отримує значення наступного елемента колекції на кожному кроці виконання циклу **foreach**. А *ім’я_колекції* позначає колекцію, що циклічно переглядається. Отже, тип змінної циклу повинен відповідати типу елемента колекції. Крім того, тип може позначатися ключовим словом **var**. В цьому випадку компілятор визначає тип змінної циклу, виходячи з типу елемента колекції.

Оператор циклу **foreach** діє таким чином. Коли цикл починається, перший елемент колекції вибирається і присвоюється змінній циклу. На кожному наступному кроці ітерації вибирається наступний елемент колекції, який зберігається в змінній циклу. Цикл завершується, коли всі елементи колекції виявляться переглянутими. Отже, оператор **foreach** циклічно опитує колекції по окремих її елементів від початку і до кінця.

Слід, однак, мати на увазі, що змінна циклу в операторі **foreach** служить тільки для читання. Це означає, що, присвоюючи цій змінній нове значення, не можна змінити вміст масиву.

Незважаючи на те що цикл **foreach** повторюється до тих пір, поки не будуть переглянуті всі елементи колекції, його можна завершити достроково, скориставшись оператором break.

Оператор **foreach** допускає циклічне звернення до масиву тільки в певному порядку: від початку і до кінця масиву, тому його застосування здається, на перший погляд, обмеженим. Але насправді це не так. У великій

кількості алгоритмів, найпоширенішим з яких є алгоритм пошуку, потрібно саме такий механізм.

Розглянемо приклад програми, в якій показана робота з колекцією об'єктів класу *Lamp*.

```
using System;
using System.Collections.Generic;

namespace ClassElementTest
{
    class Lamp
    {
        private string _name;
        private double _power;

        public double Power
        {
            get { return _power; }
            set
            {
                if (value >= 0)
                {
                    _power = value;
                }
            }
        }

        public double Voltage { get; set; }

        public double Current
        {
            get { return _power / Voltage; }
        }

        public Lamp(string nameValue, double powerValue, double voltageValue)
        {
            _name = nameValue;
            Power = powerValue;
            Voltage = voltageValue;
        }

        public override string ToString()
        {
            return String.Format("Лампа {0}: потужність - {1} Вт, напруга -
{2} В, струм - {3:n3} А", _name, Power, Voltage, Current);
        }
    }
    class Program
    {
        static void Main(string[] args)
        {
            //Створити список
            var lampList = new List<Lamp>();
```

```
            //Додати елементи в список
            lampList.Add(new Lamp("СЛ-21", 15, 220));
            lampList.Add(new Lamp("ПЛ-345", 75, 220));
            lampList.Add(new Lamp("РПС", 6, 12));
            lampList.Add(new Lamp("ДМ45", 45, 24));
            lampList.Add(new Lamp("Е100", 100, 220));
            //Вивести список на екран використовуючи цикл for
            Console.WriteLine("Список ламп за допомогою циклу for");
            for (int i = 0; i < lampList.Count; i++)
            {
                Console.WriteLine(lampList[i]);
            }
            //Вивести список на екран використовуючи цикл for
            Console.WriteLine("Список ламп за допомогою циклу foreach");
            foreach (var lamp in lampList)
            {
                Console.WriteLine(lamp);
            }
            Console.ReadLine();
        }
    }
}
```

У розглянутій програмі є все-таки один не зовсім очевидний недолік: створену колекцію *lampList* не можна відсортувати. Справа в тому, що в класах **ArrayList і List<T>** відсутні засоби для порівняння об'єктів користувацького типу даних. Для виходу з цього становища в класі можна реалізувати інтерфейс **IComparable**, в якому визначається метод порівняння об'єктів даного класу. Якщо потрібно впорядкувати об'єкти, що зберігаються в узагальненій колекції, то для цієї мети доведеться реалізувати узагальнений варіант інтерфейсу **IComparable<T>**. У цьому інтерфейсі визначається наведена нижче узагальнена форма методу **CompareTo()**:

```
int CompareTo(T other)
```

У методі CompareTo () поточний об'єкт порівнюється з іншим об'єктом other. Для сортування об'єктів по наростаючій конкретна реалізація даного методу повинна повертати нульове значення, якщо значення порівнюваних об'єктів рівні; додатнє – якщо значення поточного об'єкта більше, ніж у об'єкту other; і від'ємне – якщо значення поточного об'єкта менше, ніж у іншого об'єкта other.

Реалізуємо метод **CompareTo()** для класу *Lamp*, виконавши порівняння ламп за потужністю. Тобто будемо вважати, що більшою є лампа у якої більша потужність.

```
using System;
using System.Collections.Generic;

namespace ClassElementTest
{
    class Lamp:IComparable<Lamp>
    {
        private string _name;
        private double _power;

        public double Power
        {
            get { return _power; }
            set
            {
                if (value >= 0)
                {
                    _power = value;
                }
            }
        }

        public double Voltage { get; set; }

        public double Current
        {
            get { return _power / Voltage; }
        }

        public Lamp(string nameValue, double powerValue, double voltageValue)
        {
            _name = nameValue;
            Power = powerValue;
            Voltage = voltageValue;
        }

        public int CompareTo(Lamp other)
        {
            if (Power==other.Power)
            {
                return 0;
            }
            else if (Power>other.Power)
            {
                return 1;
            }
            else
            {
                return -1;
            }
        }
```

```
        public override string ToString()
        {
            return String.Format("Лампа {0}: потужність - {1} Вт, напруга -
{2} В, струм - {3:n3} А",
                _name, Power, Voltage, Current);
        }
    }
    class Program
    {
        static void Main(string[] args)
        {
            //Створити список
            var lampList = new List<Lamp>();
            //Додати елементи в список
            lampList.Add(new Lamp("СЛ-21", 15, 220));
            lampList.Add(new Lamp("ПЛ-345", 75, 220));
            lampList.Add(new Lamp("РПС", 6, 12));
            lampList.Add(new Lamp("ДМ45", 45, 24));
            lampList.Add(new Lamp("Е100", 100, 220));
            //Список до сортування
            Console.WriteLine("Список ламп до сортування");
            foreach (var lamp in lampList)
            {
                Console.WriteLine(lamp);
            }
            //Відсортуємо список
            lampList.Sort();
            Console.WriteLine("Відсортований список ламп");
            foreach (var lamp in lampList)
            {
                Console.WriteLine(lamp);
            }
            Console.ReadLine();
        }
    }
}
```

Практичне заняття №4. Файлове введення та виведення

У середовищі .NET Framework передбачені класи для організації введення-виведення в файли. На рівні операційної системи файли складаються з байтів, тому існують відповідні методи для введення і виведення байтів в файли. Крім того, існують спеціальні операції символьного введення-виведення в файл, які застосовуються при обробці тексту.

Для створення байтового потоку, пов’язаного з файлом, служить клас **FileStream**. Клас **FileStream** визначено в просторі імен **System.IO**. Тому на початку будь-якої програми, що його використовує клас **FileStream**, додається такий рядок коду:

```
using System.IO;
```

Для відкриття файлу створюється об'єкт класу **FileStream**. В цьому класі визначено кілька конструкторів. Нижче наведено найпоширеніший серед них:

```
FileStream (string шлях, FileMode режим)
```

де *шлях* позначає ім'я файлу, включаючи повний шлях до нього; а *режим* порядок відкриття файлу, де вказується одне зі значень, що перерахуванні табл. 6.1. Як правило, цей конструктор відкриває файл для доступу з метою читання або запису. Винятком з цього правила є відкриття файлу в режимі **FileMode.Append**, коли файл стає доступним тільки для запису.

Значення режимів відкриття файлів FileMode

Значення	Опис
FileMode.Append	Додає дані в кінець файлу
FileMode.Create	Створює новий вихідний файл. Існуючий файл з таким же ім'ям буде знищено
FileMode.CreateNew	Створює новий вихідний файл. Файл з таким же ім'ям не повинен існувати

FileMode.Open	Відкриває існуючий файл
FileMode.OpenOrCreate	Відкриває файл, якщо він існує. В іншому випадку створює новий файл
FileMode.Truncate	Відкриває існуючий файл, але скорочує його довжину до нуля

Якщо спроба відкрити файл виявляється невдалою, то генерується помилка:

- якщо файл не можна відкрити тому що він не існує, генерується помилка **FileNotFoundException**;
- якщо файл не можна відкрити через помилки введення-виведення, генерується помилка **IOException**.

До числа інших помилок, які можуть бути згенеровані при відкритті файлу, відносяться такі: **ArgumentNullException** (вказано порожнє ім'я файлу), **ArgumentException** (вказано невірне ім'я файлу), **ArgumentOutOfRangeException** (вказана невірний режим), **SeaurityException** (у користувача немає прав доступу до файлу), **PathTooLongException** (занадто довге ім'я файлу або шлях до нього), **NotSupportedException** (в імені файлу вказано пристрій, який не підтримується), а також **DirectoryNotFoundException** (вказано невірний каталог).

По завершенні роботи з файлом його слід закрити, викликавши метод **Close()**.

У класі **FileStream** визначені два методу для читання байтів з файлу: **ReadByte()** і **Read()**. Так, для читання одного байта з файлу використовується метод **ReadByte()**, загальна форма якого наведена нижче:

```
int ReadByte()
```

Коли цей метод викликається, з файлу зчитується один байт, який потім повертається у вигляді цілого значення.

Для читання блоку байтів з файлу служить метод **Read()**, загальна форма якого виглядає так.

```
int Read(byte [] array, int offset, int count)
```

У методі **Read()** робиться спроба зчитати кількість *count* байтів в масив *array*, починаючи з елемента *array*[*offset*]. Метод повертає кількість байтів, успішно зчитаних з файлу. Якщо ж виникає помилка введення-виведення, то генерується виключення **IOException**.

Для запису байта в файл служить метод **WriteByte()**. Нижче наведена його найпростіша форма.

```
void WriteByte(byte value)
```

Цей метод виконує запис в файл байта, що позначається параметром *value*.

Для запису в файл цілого масиву байтів може бути викликаний метод **Write()**. Нижче наведена його загальна форма.

```
void Write(byte [] array, int offset, int count)
```

У методі **Write()** робиться спроба записати в файл кількість *count* байтів з масиву *array*, починаючи з елемента *array*[*offset*]. Метод повертає кількість байтів, успішно записаних в файл.

По завершенні виведення в файл, його слід закрити за допомогою методу **Close()**.

Незважаючи на те, що файли часто обробляються побайтово, для цього можна скористатися також символьними потоками. Перевага символьних потоків полягає в тому, що вони оперують символами безпосередньо в Unicode. Якщо потрібно зберегти текст, то для цього найкраще підійдуть саме символьні потоки. В цілому, для виконання операцій символьного введення-виведення в файли використовуються класи **StreamReader** та **StreamWriter**.

Для створення символьного потоку виведення досить укласти об'єкт класу **Stream**, наприклад **FileStream**, в оболонку класу **StreamWriter**. У класі

StreamWriter визначено кілька конструкторів. Нижче наведено найпоширеніший серед них:

```
StreamWriter(Stream потік)
```

де *потік* позначає ім'я відкритого потоку. Цей конструктор генерує виняток **ArgumentException**, якщо потік не відкритий для виведення, а також виключення **ArgumentNullException**, якщо потік виявляється порожнім.

У деяких випадках зручніше відкривати файл засобами самого класу **StreamWriter**. Для цього служить один з наступних конструкторів:

```
StreamWriter(string шлях)
StreamWriter(string шлях, bool append)
```

де *шлях* – це ім'я файлу, включаючи повний шлях до нього. Якщо в другій формі цього конструктора значення параметра *append* рівне **true**, то дані приєднуються в кінець існуючого файлу. В іншому випадку ці дані перезаписують вміст зазначеного файлу. Але незалежно від форми конструктора файл створюється, якщо він не існує.

Нижче наведено простий приклад програми виведення на диск опису об’єкту класу *Lamp* у вигляді текстових рядків, як зберігаються в файлі test.txt. Для символьного виводу в файл в цій програмі використовується об'єкт класу FileStream, укладений в оболонку класу StreamWriter.

```
using System;
using System.IO;

namespace ClassElementTest
{
    class Lamp
    {  ...  }

    class Program
    {
        static void Main(string[] args)
        {
            var sampleLamp = new Lamp("СЛ-21", 15, 220);
            var fout=new StreamWriter("text.txt");
            fout.WriteLine(sampleLamp);
            fout.Close();
        }
    }
```

```
}
```

Для створення символьного потоку введення досить укласти байтовий потік в оболонку класу **StreamReader**. У класі **StreamReader** визначено кілька конструкторів. Найбільш часто використовується конструктор:

```
StreamReader(Stream потік)
```

де *потік* позначає ім'я відкритого потоку. Цей конструктор генерує виняток **ArgumentNullException**, якщо потік виявляється порожнім, а також виключення **ArgumentException**, якщо потік не відкритий для введення.

Іноді файл простіше відкрити, використовуючи безпосередньо клас **StreamReader**, аналогічно класу **StreamWriter**. Для цієї мети служить наступний конструктор:

```
StreamReader(string шлях)
```

де *шлях* – це ім'я файлу, включаючи повний шлях до нього. Вказаний файл повинен існувати. В іншому випадку генерується виняток **FileNotFoundException**. Якщо шлях виявляється порожнім, то генерується виключення **ArgumentNullException**.

Нижче наведено простий приклад програми зчитування з диску текстових рядків, як зберігаються в файлі *text.txt*.

```
using System;
using System.IO;
class Program
{
    static void Main()
    {
        var fin = new StreamReader(@"D:\text.txt");
        while (!fin.EndOfStream)
        {
            var s = fin.ReadLine();
            Console.WriteLine(s);
        }
        fin.Close();
    }
}
```

Зверніть увагу на те, як в цій програмі визначається кінець файлу. Доступна для читання властивість **EndOfStream** має логічне значення **true**,

коли досягається кінець потоку, в іншому випадку – логічне значення **false**. Отже, властивість **EndOfStream** можна використовувати для відстеження кінця файлу.

У середовищі .NET Framework також визначено клас **File**, який є корисним для роботи з файлами, оскільки він містить статичні методи, що виконують типові операції над файлами. Зокрема, в класі **File** є методи для копіювання та переміщення, шифрування і розшифрування, видалення файлів, а також для отримання і завдання інформації про файлах, включаючи відомості про їхнє існування, часу створення, останнього доступу і різні атрибути файлів (тільки для читання, прихованих і ін.). Крім того, в класі **File** є ряд зручних методів для читання з файлів і записи в них, відкриття файлу і отримання посилання типу **FileStream** на нього.

Ряд методів для роботи з файлами визначено також в класі **FileInfo**. Цей клас відрізняється від класу **File** дуже важливою перевагою: для операцій над файлами він надає методи екземпляра і властивості, а не статичні методи. Тому для виконання декількох операцій над одним і тим же файлом краще скористатися класом **FileInfo**.

У C# є можливість зберігати не тільки дані примітивних типів, але і об'єкти. Термін серіалізація описує процес збереження (і, можливо, передачі) стану об'єкта в потоці (наприклад, файловому потоці і потоці в пам'яті). Послідовність даних, що зберігається містить всю інформацію, необхідну для реконструкції (або десеріалізації) стану об'єкта з метою подальшого використання. Застосовуючи цю технологію, дуже просто зберігати великі обсяги даних (в різних форматах) з мінімальними зусиллями. У багатьох випадках збереження даних програми з використанням серіалізації виливається в код меншого обсягу, ніж застосування класів для читання/запису з простору імен **System.IO**.

Щоб зробити об'єкт доступним для серіалізації, необхідно позначити клас (або структуру) атрибутом **[Serializable]**. Атрибути – це додаткові

відомості про клас, які зберігаються в його метаданих. Якщо існують поля, які не повинні (або не можуть) брати участь в серіалізації, можна помітити такі поля атрибутом **[NonSerialized]**.

Об'єкти можна зберігати в одному з таких форматів: бінарному, SOAP або у вигляді XML-файла. Перераховані можливості представлені такими класами: **BinaryFormatter, SoapFormatter, XmlSerializer.**

Тип **BinaryFormatter** серіалізує стан об'єкта в потік, використовуючи компактний бінарний формат. Цей тип визначений в просторі імен **System.Runtime.Serialization.Formatters.Binary**. Таким чином, щоб отримати доступ до цього типу, необхідно вказати наступну директиву **using**:

```
using System.Runtime.Serialization.Formatters.Binary;
```

Тип SoapFormatter зберігає стан об'єкта у вигляді повідомлення SOAP. Цей тип визначений в просторі імен **System.Runtime.Serialization.Formatters.Soap**, що знаходиться в окремій збірці. Тому для форматування об'єктів в повідомлення SOAP необхідно спочатку встановити посилання на *System.Runtime.Serialization.Formatters.Soap.dll*, використовуючи діалогове вікно *Add Reference* (Додати посилання) в Visual Studio і потім вказати наступну директиву **using**:

```
using System.Runtime.Serialization.Formatters.Soap;
```

Для збереження об'єктів в документі XML передбачений тип **XmlSerializer**. Щоб використовувати цей тип, потрібно вказати директиву **using**:

```
using System.Xml.Serialization;
```

Двома ключовими методами типу **BinaryFormatter** є: **Serialize()** – зберігає об'єкт в зазначений потік у вигляді послідовності байтів; **Deserialize()** – перетворює збережену послідовність байтів в об'єкт.

Припустимо, що після створення екземпляра класу і модифікації деяких даних стану потрібно зберегти цей екземпляр у файлі *.dat. Почати слід з створення самого файлу *.dat. Для цього можна створити екземпляр типу **System.IO.FileStream**. Потім потрібно буде створити екземпляр **BinaryFormatter** і передати йому **FileStream** і об'єкт для збереження.

Для прикладу розглянемо додаток в якому реалізована серіалізація об’єктів типу Lamp.

```
using System;
using System.IO;
using System.Runtime.Serialization.Formatters.Binary;

namespace ClassElementTest
{
    [Serializable]
    class Lamp
    {   ...   }
    class Program
    {
        static void Main(string[] args)
        {
            var sampleLamp = new Lamp("СЛ-21", 15, 220);
            var binFormat = new BinaryFormatter();
            var fStream = new FileStream("Lamp.dat", FileMode.Create);
            binFormat.Serialize(fStream, sampleLamp);
            fStream.Close();
        }
    }
}
```

Після виконання програми можна переглянути вміст файлу *Lamp.dat* в папці *bin\Debug* поточного проекту.

Тепер припустимо, що необхідно прочитати збережений об'єкт із двійкового файлу назад в змінну. Після відкриття файлу *Lamp.dat* (за допомогою методу **File.OpenRead()**) можна викликати метод **Deserialize()** класу **BinaryFormatter**. **Deserialize()** повертає об'єкт загального типу **System.Object**, тому необхідно застосувати явне приведення, як показано нижче:

```
using System;
using System.IO;
using System.Runtime.Serialization.Formatters.Binary;
```

```
namespace ClassElementTest
{
    [Serializable]
    class Lamp
    {  ...  }
    class Program
    {
        static void Main(string[] args)
        {
            var dataFile = File.OpenRead(@"D:\Lamp.dat");
            var binFormat = new BinaryFormatter();
            var sampleLamp = (Lamp)binFormat.Deserialize(dataFile);
            Console.WriteLine(sampleLamp);
        }
    }
}
```

Серіалізация об'єктів з використанням XmlSerializer може використовуватися для збереження стану заданого об'єкта у вигляді чистої XML-розмітки. XML-документ складається із тексту, і придатний до читання людиною. Розглянемо наступний код:

```
using System;
using System.IO;
using System.Runtime.Serialization.Formatters.Binary;
using System.Xml.Serialization;

namespace ClassElementTest
{
    [Serializable]
    public class Lamp
    {   ...   }
    class Program
    {
        static void Main(string[] args)
        {
            {
                var sampleLamp = new Lamp("СЛ-21", 15, 220);
                var xmlFormat = new XmlSerializer(typeof (Lamp));
                var xmlFile = new FileStream("Lamp.xml", FileMode.Create);
                xmlFormat.Serialize(xmlFile, sampleLamp);
                xmlFile.Close();
            }
            {
                var xmlFile = File.OpenRead(@"D:\Lamp.xml");
                var xmlFormat = new XmlSerializer(typeof(Lamp));
                var sampleLamp = (Lamp)xmlFormat.Deserialize(xmlFile);
                Console.WriteLine(sampleLamp);
            }
        }
    }
```

```
}
```

Ключова відмінність, від прикладу з **BinaryFormatter** полягає в тому, що тип **XmlSerializer** вимагає вказівки інформації про клас, який необхідно серіалізувати. У створеному файлі XML знаходяться показані нижче дані XML:

```
<?xml version="1.0"?>
<Lamp xmlns:xsd="http://www.w3.org/2001/XMLSchema"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
  <Power>15</Power>
  <Voltage>220</Voltage>
</Lamp>
```

Клас **XmlSerializer** вимагає, щоб всі серіалізовані типи об'єктів підтримували стандартний конструктор (без параметрів).

Особливістю сереалізації колекцій об'єктів, є те, що більшість типів з просторів імен **System.Collections** і **System.Collections.Generic** вже позначені атрибутом **[Serializable]**. Таким чином, щоб зберегти сукупність об'єктів, можна просто додати їх в колекцію (таку як звичайний масив, **ArrayList** або **List<T>**) і серіалізувати даний об'єкт в бажаний потік (файл).

Практичне заняття №5. Розв’язання СЛАР методом Гауса.

Розглянемо підхід до розв'язання систем лінійних алгебраїчних рівнянь виду

$$\begin{cases} a_{11}x_1 + a_{12}x_2 + \cdots + a_{1n}x_n = b_1 \\ \cdots \\ a_{m1}x_1 + a_{m2}x_2 + \cdots + a_{mn}x_n = b_m \end{cases} \quad (1)$$

або у матричній формі запису

$$[A][X]=[B] \quad (2)$$

Розвязання можливе прямими методами класичної математики, тобто їх точне розв'язання можна отримати за визначену певну кількість арифметичних дій. В системі (1) $a_{i,j}\ (i = 1..m, j = 1..n)$ – відомі числові коефіцієнти, b_i- відомі праві частини, а x_i - обчислювані невідомі величини.

У практичних інженерних задачах матриця [A] майже завжди квадратна *(m=n)* та неособлива, тобто її визначник не дорівнює нулю. У разі, коли *m>n* маємо перевизначену систему, що не має розв'язку, а якщо *m<n,* то система недовизначена і має безліч розв'язків [X], що задовольняють її. Перший випадок свідчить про помилки при створенні математичної моделі явища, другий може мати місце в оптимізаційних задачах. Отже, далі вважатимемо, що матриця [A] квадратна та неособлива, в наслідок чого її завжди можна обернути, одержавши матрицю $[A]^{-1}$, а далі, помноживши ліву частину рівняння (2) на цю матрицю, отримати рівняння:

$$[X]=[A^{-1}][B] \quad (3)$$

яке дозволяє визначити невідомі X_i прямими обчисленнями.

$$[A]^{-1}[A][X]=[A]^{-1}[B]$$

$$[A]^{-1}[A]=[E]$$

$$[E][X]=[A]^{-1}[B]$$

$$[E][X]=[X]$$

$$[X]=[A]^{-1}[B]$$

Тут [E] - одинична матриця, тобто матриця, у якої головна діагональ має одиниці, а решта елементів нульові.

Метод Гауса

З прямих методів розв'язування систем лінійних алгебраїчних рівнянь найчастіше використовується метод Гауса та його модифікації. Обернена матриця $[A]^{-1}$ при цьому в явному вигляді не обчислюється. Процес розв'язання методом Гауса складається з двох чітко розмежованих етапів - прямого та зворотного «ходів». Етап прямого ходу є перетворення початкової системи рівнянь (1) в еквівалентну, що має наступний вигляд:

$$\begin{cases} x_n + a_{12}x_2 + a_{13}x_3 + \cdots + a_{1n}x_n = b_1 \\ x_2 + a_{23}x_3 + \cdots + a_{2n}x_n = b_2 \\ \cdots \\ x_n = b_n \end{cases} \qquad (4)$$

Здійснити таке перетворення можна послідовним повторенням *n-1* раз наступних однотипних дій:

Коефіцієнти першого рівняння ділять на коефіцієнт a_{11}

$$a'_{12} = \frac{a_{12}}{a_{11}};\ a'_{13} = \frac{a_{13}}{a_{11}};\ \ldots;\ a'_{1n} = \frac{a_{1n}}{a_{11}};\ b'_1 = \frac{b_1}{a_{11}}$$

В рівняння номер 2, 3,…,n замість невідомого x_1, підставляється його формульне значення, одержане з першого рівняння

$$x_1 = b_1 - a_{12}x_2 - a_{13}x_3 - \cdots - a_{1n}x_n$$

після чого зводяться подібні члени. Це по суті досягається коригуванням коефіцієнтів цих рівнянь по формулам:

$$a'_{i,k} = a_{i,k} - a_{i,1}a_{1,k}$$

$$b'_i = b_i - a_{i,1}b_1\ (i = 2..n, k = 2..n)$$

В результаті перше рівняння системи отримує вигляд, потрібний для (4), а в решті рівнянь зникає невідоме x_1. Ці *n-1* рівняння розглядаються як початкова система й для них повторюються дії 1-2, що дозволяє отримати друге рівняння системи (4).

Послідовне повторення вказаних дій до того часу, поки система (1) не перетвориться в одне рівняння, забезпечує одержання перетвореної системи (4). Зауважимо, що для того щоб потрібні для перетворення дії було можливо виконати, необхідно, щоб кожний діагональний («ведучий») коефіцієнт $a_{i,j}$ не дорівнював нулю. Це можна забезпечити відповідною перестановкою у разі необхідності рівнянь системи (1). Більше того, якщо діагональні, коефіцієнти по модулю гарантовано більші будь-якого іншого коефіцієнта рівняння $(|a_{i,j}| > |a_{i,k}|, i,k = 1..n, i \neq k)$, то похибки обчислень, що виникають внаслідок округлень та втрати останніх цифр, суттєво зменшуються. При перетасуванні рівнянь початкової системи по вказаному критерію вибору діагональних елементів домінуючими виникає модифікація метода Гауса «з вибором головного елемента».

Другий - кінцевий етап метода Гауса - Його «зворотній хід», веде до обчислення значень шуканого вектора [X] в зворотному циклі по перетвореній до вигляду (4) системі:

З останнього рівняння обчислюється x_n;

$$x_n = b_n$$

Одержане числове значення x_n підставляється в усі 1..(n-1) рівняння та зводяться подібні члени. Кількість рівнянь та невідомих *x* при цьому зменшується на одиницю, а останнє рівняння системи набуває вигляду $x_{n-1}=b_{n-1}$. Далі кроки повторюються доти, поки не буде отримано числове значення невідомого x_1.

Практичне заняття №6. Робота з формами

Windows Forms дозволяє розробляти інтелектуальні клієнти. **Інтелектуальний клієнт** - це програма з багатим графічним інтерфейсом, проста в розгортанні і оновленні, здатна працювати при наявності або відсутності підключення до Інтернету, яка використовує більш безпечний доступ до ресурсів на локальному комп'ютері в порівнянні з традиційними додатками Windows.

Windows Forms є технологією інтелектуальних клієнтів для .NET Framework; це набір керованих бібліотек для спрощення реалізації програмних задач, наприклад читання і запис в файлову систему. За допомогою середовища розробки типу Visual Studio можна створювати додатки Windows Forms, які відображають інформацію, запитують введення користувачів і обмінюються даними з віддаленими комп'ютерами по мережі.

У Windows Forms форма є видимою поверхнею, на якій відображається інформація для користувача. Зазвичай додаток Windows Forms будується шляхом розміщення елементів управління на формі і написанням коду для реагування на дії користувача, такі як клацання миші або натискання клавіш. Елемент управління - це окремий елемент інтерфейсу, призначений для відображення або введення даних.

При виконанні користувачем якої-небудь дії з формою або одним з її елементів управління, створюється подія. Додаток реагує на ці події за допомогою коду і обробляє події при їх виникненні. Додаткові відомості див Створення обробників подій в Windows Forms.

Windows Forms включає широкий набір елементів управління, які можна додавати на форми: текстові поля, кнопки, списки, що розкриваються, перемикачі та навіть веб-сторінки. Список всіх елементів управління, які можна використовувати в формі, см Елементи управління для використання в формах Windows Forms. Якщо існуючий елемент управління не задовольняє потребам, в Windows Forms можна створити власні настроюються елементи управління за допомогою класу UserControl.

До складу Windows Forms входять елементи призначеного для користувача інтерфейсу з розширеними функціями, відповідними можливостям потужних додатків, таких як Microsoft Office. Використовуючи елементи управління ToolStrip і MenuStrip, можна створювати панелі інструментів і меню, що містять текст і малюнки, що відображають підміню і містять в собі інші елементи управління, такі як текстові поля і поля з списком, що випадає.

За допомогою конструктора Windows Forms Visual Studio, що підтримує перетягування, можна легко створювати додатки Windows Forms: Досить виділити елемент керування курсором і помістити його на потрібне місце на формі. Конструктор надає такі кошти, як лінії сітки і "прив'язка ліній" для подолання труднощів вирівнювання елементів управління. І в разі використання Visual Studio або компіляції з командного рядка можна використовувати елементи управління FlowLayoutPanel, TableLayoutPanel і SplitContainer для створення просунутих розміток форми за мінімальний час і з мінімальними зусиллями.

System.Windows.Forms - простір імен

Простір імен System.Windows.Forms містить класи для створення додатків Windows, які дозволяють Найбільш ефективно використовувати Розширені можливості призначеного для користувача інтерфейсу, які надаються операційною системою Microsoft Windows.

У наступній таблиці наведено згруповані за категоріями класи, що відносяться до простору імен System.Windows.Forms.

Категорія класів	Докладні відомості
Елементи управління, призначені для користувача елементи управління і форми	Більшість класів в просторі імен **System.Windows.Forms** є похідними від класу Control .Клас Control надає основні функціональні можливості для всіх елементів управління, які відображаються в Form .

Меню і панелі інструментів	Windows Forms включає широкий набір класів, які дозволяють створювати призначені для користувача панелі інструментів і меню, що відрізняються сучасним вигляд і поведінкою. ToolStrip , MenuStrip , ContextMenuStrip і StatusStrip дозволяють створювати панелі інструментів, терміни меню, контекстні меню і терміни стану, відповідно.
Елементи управління	Простір імен **System.Windows.Forms** надає велику кількість класів елементів управління, які дозволяють створювати призначені для користувача інтерфейси з розширеним можливостями.
Макет	Кілька принципових класів в Windows Forms допомагають контролювати розташування елементів управління на Отображаемое поверхні наприклад в формі або елементі управління. На панелі FlowLayoutPanel послідовно розміщені всі елементи управління, які вона містить.
Дані і прив'язка даних	Windows Forms забезпечує розширену архітектуру для прив'язування до таких джерел даних, як бази даних і XML-файли. Елемент управління DataGridView надає настроюється таблицю для відображення даних і дозволяє налаштовувати формат осередків, термін, стовпців і кордонів.
Компоненти	Крім елементів управління простір імен **System.Windows.Forms** надає інші класи, які НЕ є похідними від класу Control , але також забезпечують візуальні функції для додатків Windows.Такі класи, як ToolTip і ErrorProvider , розширюють можливості або Предоставляютя відомості користувачам.

У просторі імен **System.Windows.Forms** є ряд класів, що забезпечують підтримку вищезазначених класів.Прикладами підтримує класів можуть

служить перерахування, класи аргументів подій і Делегати, які використовуються подіями в елементах управління і компонентах.

Створення форми Windows Forms

1. Запустіть Visual Studio.
2. Створіть додаток Windows з ім'ям HelloWorld.
3. З **панелі елементів** перетягнути в форму елемент управління **Button.**
4. Виділити його клацанням миші.У вікні "Властивості" надайте властивості Text значення "Say Hello".

Написання коду програми

1. Двічі Клацніть кнопку, щоб додати оброблювач подій Click .Відкриється редактор коду, при цьому положення курсора виявиться всередині обробника подій.
2. Вставте цей код:

```
MessageBox.Show ( "Hello, World!");
```

Тестування додатка

1. Натисніть клавішу F5, щоб запустить додаток.
2. Коли додаток запущено, натисніть кнопку і перевірте, чи відображається фраза "Hello, World!"
3. Закрийте форму Windows Forms, щоб повернутися в Visual Studio.

Створення обробника подій за допомогою конструктора

1. Клацніть форму або елемент керування, для якого потрібно створити обробник подій.
2. У вікні Властивості натисніть кнопку Події ().
3. У списку доступних подій Клацніть подія, для якого потрібно створити обробник подій.
4. В поле праворуч від імені події введіть ім'я обробника і натисніть клавішу ENTER.

З'явиться Редактор коду з кодом для форми; метод обробника подій створюється в коді за аналогією з кодом в наступному прикладі:

```
private void StartProcess (object sender, System.EventArgs e) {
   // Add event handler code here.
}
```

5. Додати Відповідний код в обробник подій.

Практичне заняття №7. Робота з елементами форм

В ході розробки і зміни призначеного для користувача інтерфейсу додатків Windows Forms потрібно додавати, вирівнювати і розміщувати елементи управління. Елементи управління - це об'єкти, які знаходяться всередині об'єктів форми. Кожен тип елемента управління має власний набір властивостей, методів і подій, що відповідають певному призначенню. З елементами управління можна працювати в конструкторі або додавати їх динамічно під час виконання з допомогою коду.

Функціональна класифікація елементів управління Windows Forms

У Windows Forms існують елементи управління і компоненти, що виконують ряд функцій. У наступній таблиці представлений список компонентів і елементів управління Windows Forms відповідно до основною функцією. Крім того, якщо для виконання однієї і тієї ж функції служать кілька елементів управління, то рекомендовані елементи управління перераховані із зазначенням використовувалися раніше застарілих елементів управління. Застарілі елементи управління також перераховані в окремій таблиці; поруч з кожним таїмо елементом управління вказано новий елемент, що прийшов на зміну застаріле.

Функція	Елемент управління	Опис
Відображення даних	DataGridView	Елемент управління DataGridView надає настроюється таблицю для відображення даних.Клас DataGridView забезпечує настройку осередків, термін, стовпців і меж таблиці. **Примітка** елемент управління DataGridView підтримує ряд простих і складних функцій, відсутніх в елементі управління DataGrid .Додаткові відомості див. В розділі Відмінності елементів управління DataGridView і DataGrid в Windows Forms .
Прив'язка даних і переміщення	BindingSource	Спрощує прив'язку елементів управління у формі до даних завдяки засобам управління

		грошовими одиницями, повідомлення про зміни і т.д.
	BindingNavigator	Надає інтерфейс, подібний панелі інструментів, для переходів по формі і управління даними.
редагування тексту	TextBox	Відображає текст, введений в режимі розробки, Який користувачі можуть змінювати під час виконання або за допомогою програмних засобів.
	RichTextBox	Дозволяє представлять текст в текстових форматі або в форматі RTF.
	MaskedTextBox	Обмежує формат даних, що вводяться користувачем.
Відображення інформації (тільки для читання)	Label	Відображає текст, недоступний для безпосереднього редагування користувачем.
	LinkLabel	Відображає текст у вигляді веб-посилання і створює подія при клацанням тексту. Як правило, текст є посиланням на інше вікно або на веб-вузол.
…	…	…

Додавання елементів керування в форми Windows Forms.

1. Відкрийте форму.
2. В **панелі елементів** Клацніть елемент керування, Який потрібно додати в форму.
3. Клацніть місце в формі, де повинен розташовуватися лівий верхній кут елемента керування, а потім перетягнути покажчик миші на місце, в якому повинен розташовуватися правий нижній кут елемента керування.

Елемент управління додається на форму у зазначеній місце з Зазначеними розмірами.

Практичне заняття №8. Малювання на формах.

Простір імен System.Drawing забезпечує доступ до функціональних можливостей графічного інтерфейсу GDI +. Простору імен System.Drawing.Drawing2D, System.Drawing.Imaging, і System.Drawing.Text забезпечують додаткові функціональні можливості.

Клас Graphics надає методи малювання на пристрої відображення. Такі класи, як Rectangle і Point інкапсулюють елементи GDI +. Клас Pen використовується для малювання ліній і кривих, а класи, похідні від абстрактного класу Brush, використовуються для заливки фігур.

Клас	Опис
Bitmap	Інкапсулює точковий малюнок GDI +, що складається з даних точок графічного зображення і атрибутів малюнка. об'єкт Bitmap використовується для роботи з зображеннями, які визначаються даними точок.
Brush	Визначає об'єкти, які використовуються для заливки всередині графічних фігур, таких як прямокутники, еліпси, кола, багатокутника і доріжки.
Brushes	Кисті для кожного зі стандартних кольорів. Цей клас бути не може бути успадкований.
BufferedGraphics	Надає графічний буфер для подвійний буферизації.
BufferedGraphicsContext	Надає методи створення графічних буферів, які можуть вживатися для подвійний буферизації.
BufferedGraphicsManager	Надає доступ до об'єкта основного контексту буферизує графіки для домену додатки.
ColorConverter	Перетворює кольори одного типу даних в інший. Доступ до даного класу здійснюється за допомогою об'єкта TypeDescriptor .

ColorTranslator	Здійснює переклад кольорів в структури GDI + Color і з них.Цей клас бути не може бути успадкований.
Font	Визначає конкретний формат тексту, включаючи Нарис шрифту, його розмір і атрибути стилю. Цей клас бути не може бути успадкований.
FontConverter	перетворює об'єкти Font з одного типу даних в інший.
FontConverter. FontNameConverter	Інфраструктура. FontConverter.FontNameConverter - перетворювач типу, Який використовується для перетворення імені шрифту в інші різні уявлення і назад.
FontConverter. FontUnitConverter	Інфраструктура. Перетворює одиниці шрифту в інші типи одиниць і назад.
FontFamily	Визначає групу гарнітур зі схожим базовим конструктором і певними відмінностями в стилі. Цей клас бути не може бути успадкований.
Graphics	Інкапсулює поверхню малювання GDI +. Цей клас не можна успадкувати.
…	…

Структура	Опис
CharacterRange	Визначає діапазон позицій символу в межах рядка.
Color	Являє кольору в термінах каналів альфа, червоного, зеленого і синього (ARGB).
Point	Являє впорядковану пару цілих чисел - координат X і Y, визначальну точку на двовимірної площині.
PointF	Являє впорядковану пару координат X і Y з плаваючою комою, що визначає точку на двовимірної площині.

Rectangle	Містить набір з чотирьох цілих чисел, що визначають розташування і розмір прямокутника. Для розширення функцій області Використовуйте об'єкт Region .
RectangleF	Містить набір з чотирьох цифр з плаваючою комою, що визначають розташування і розмір прямокутника. Для розширення функцій області Використовуйте об'єкт Region .
Size	Зберігає впорядковану пару цілих чисел, зазвичай ширину і висоту прямокутника.
SizeF	Містить впорядковану пару чисел з плаваючою комою, зазвичай ширину і висоту прямокутника.

Приклад який малює зелений круг на формі

```
// Подія форми Paint
 private void Form1_Paint (object sender, PaintEventArgs e) {
    // Create pen.
    Pen blackGreen = new Pen (Color.Green, 3);

    // Create rectangle for ellipse.
    Rectangle rect = new Rectangle (50, 50, 100, 100);

    // Draw ellipse to screen.
   e.Graphics.DrawEllipse (blackGreen, rect);
```

Практичне заняття №9. Побудова графіку функції

Для побудови графіка функції на форму потрібно помістити Control-елемент PictureBox. Для того щоб цей елемент візуально виділявся на області форми, рекомендується спочатку на форму помістити елемент-контейнер Panel, у якого задати властивість BorderStyle рівним FixedSingle, а вже потім помістити на нього - PictureBox. В класі PictureBox визначено важлива властивість - Graphics. За допомогою численних параметрів класу Graphics можна виводити в область, зайняту елементом управління, текстові написи, геометричні фігури і різні зображення.

Клас Graphics передається в якості параметра PaintEventArgs e для події Paint, яке відіграє дуже важливу роль в додатках Windows Forms. Воно забезпечує **перемальовування** вікна програми, як тільки в цьому виникне необхідність, наприклад, якщо поле одного вікна було тимчасово зайняте іншим вікном, або потрібно висновок другого графіка.

Тому завдання побудови графіка рекомендується виконувати, викликаючи метод Paint.

Нижче наведено фрагмент коду для виведення зображення системи координат з впровадженням методу DrawLine (малювати лінію) класу Graphics.

Спочатку оголошується клас Graphics для елемента PictureBox1 з ім'ям grpBox, задається колір пера boxPen – чорний і товщина пера – 2 і потім малюються дві перпендикулярні лінії. Лінії малюються з впровадженням примірників класу Point (крапка).Точка задається двома координатами за допомогою оператора new. У визначенні координат використовуються властивості Width (ширина) і Height (висота) елемента PictureBox1.

```
private void pictureBox1_Paint (object sender, PaintEventArgs e)
{
     Graphics grpBox = e.Graphics;
     Pen boxPen = new Pen (Color.Black, 2) grpBox.DrawLine
     (boxPen,new Point (10 10), new Point (10 pictureBox1.Height -
     10));     grpBox.DrawLine     (boxPen,new     Point     (10,
     pictureBox1.Height - 10),new Point (pictureBox1.Width - 10
     pictureBox1.Height - 10))
}
```

Розглянемо докладніше питання про те, як були обрані координати точок - початку і кінця прямих для координатних осей.

Побудова графіків на екрані монітора має наступні важливі особливості.

Перша особливість полягає в тому, що інтерфейс графічного пристрою виводить графік (за замовчуванням) в системі координат, початок якої розташовано у верхньому лівому кутку області малювання .Ось x спрямована в цьому випадку вправо, а ось y - вниз. Коли, наприклад, ми будемо розміщувати графік в елементі управління `pictureBox`, то початок координат буде розміщено саме в лівому верхньому кутку цього елемента. Для нашої задачі (число ітерацій і значення точності - позитивні величини) початок системи координат потрібно розмістити в лівому нижньому кутку. Тому початок системи координат буде в точці:

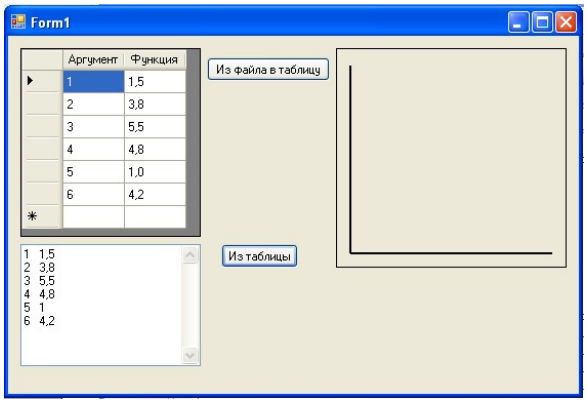
point(10, pictureBox1.Height - 10).

Константа `10` (пікселів) – це відступ від лівої і нижньої сторони `pictureBox` для розміщення написів поділів по осях. Кінцеві точки також вибираються з урахуванням цього відступу.

У загальному випадку за допомогою паралельного перенесення осей на величини `ox_pix` і `oy_pix` необхідно ввести нову систему координат, в якій ось «y» до того ж повинна бути спрямована вгору.

Друга особливість полягає в тому, що графік будується не в дійсних значеннях для функції y = f (x), які розрахує програми, а в пікселях. Тому Ми повинні підібрати масштаби `Mx` і `My` для перекладу дійсних значень змінних в пікселі з метою збільшення (або зменшення) розмірів графіка таким чином, щоб його було видно на екрані. Для змінних коду в пікселях введемо далі розширення «Pix».

На малюнку - результат роботи програми з впровадженням такого коду.



Правила побудови графіків

Щоб за пропонованою методикою можна було будувати графіки будь-яких функцій, замість дійсних значень пропонується використовувати відносні ***безрозмірні*** значення змінних.

1. Графік функції y = f (x) будується в відносних безрозмірних величинах (позначені зі штрихом) x` , y` :

x`= X / | x_max |; y` = Y / | y_max |,

де | x_max | і | y_max | - Найбільші абсолютні значення змінних x і y. Отже, будь-який графік буде знаходиться всередині прямокутника

1 = < x` <= 1;

1 = < y` <= 1

і стосуватися хоча б в одній точці двох горизонтальних прямих y`= 1 і y`= -1 і двох вертикальних x`= 1 і x`= 1.

2. Щоб Отримати дійсний значення x і y в будь-якій точці графіка, необхідно визначити відносні значення x` і y` в цій точці і помножити їх на Найбільше абсолютне значення | x_max | або | y_max |, відповідно.

3. Для отримання **координатної сітки,** в якій буде перебувати графік, кожну з двох осей координат x і y можна розбити на деяке число рівних частин, наприклад, на 20 рівних частин, і з точок розбиття провести горизонтальні і вертикальні лінії.

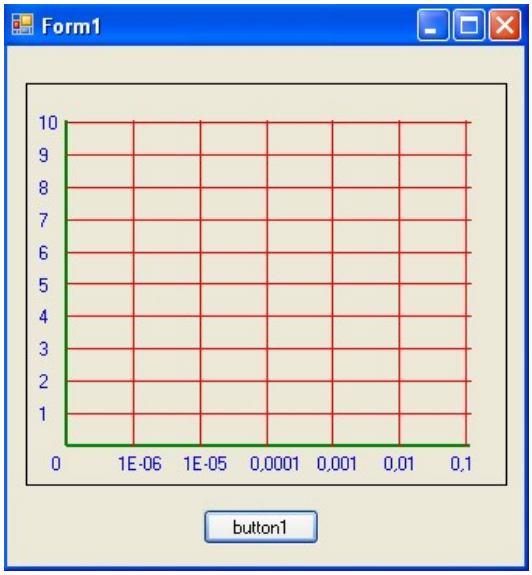
4. Поза координатної сітки слід записати цифрові або літерні позначення величин графіка.

Нижче наведено фрагмент програмного коду з коментарями, в якому побудована сітка для графіка функції, табличного значення якої отримані при обчисленнях суми нескінченного ряду (варіанта завдання).

```
public int NPoints = 6;
int indent_pix = 20; // Відступ потрібен для приміщення написів по осях public float
x_max = 0.1f; // значення x_max і x_min задані в умові завдання public float x_min =
0.00001f;
int N_step_grid_x = 7, // Щоб було 6 вертик. ліній в сітці по умові
float step_grid_x;
public int step_grid_x_pix; // крок зміни значень аргументу в пікселах public
int step_grid_y_pix; // крок зміни значень функції в пікселах
int M_x = 250;
int M_y = 210;
int x_point_end_pix;
int y_max = 5, // Значення функції y_max і y_min.Для функції ліга y_max = 5
int y_min = 1, // y_max і y_min визначаються завданням до роботі

// Оброблювач події Paint для pictureBox1
private void pictureBox1_Paint (object sender, PaintEventArgs e)
{
// Створюємо свій екземпляр класу Graphics
Graphics myGraphics = e.Graphics;
// Можна створити екземпляр Graphics інакше (див. Наступний оператор)
// Graphics myGraphics = pictureBox1.CreateGraphics ();

// визначаємо положення осей координат в пікселях
int ox_pix = indent_pix; // координата x початку координат
int oy_pix = pictureBox1.Height - indent_pix; // коорд. y початку координат
float x_point_end; // Оголошення кінцевої точки по осі x x_point_end = 1.0f; //
Значення кінцевої точки по осі x з умові

// Обчислення значення кінцевої точки осі x в пікселах с впровадженням
// масштабного коефіцієнта і з явним перетворенням типу float в тип int
```

```
x_point_end_pix = (int) x_point_end * M_x - indent_pix;
// Вибираємо зелене перо товщиною 2;
Pen greenPen_x = new Pen (Color.Green, 2);
// Задаємо координати двох граничних точок осі:
Point point1 = new Point (ox_pix, oy_pix)
Point point2 = new Point (x_point_end_pix, oy_pix)
// Будуємо лінію через дві заданий граничні точки:
myGraphics.DrawLine (greenPen_x, point1, point2)
// Для нашої задачі число горизонтальних ліній в сітці одно y_max
int N_step_grid_y = y_max;
// Крок сітки в напрямку осі "y" (висота всієї сітки дорівнює 1 одиниці)
float step_grid_y; // крок зміни значень функції
int step_grid_y_pix; // крок зміни значень функції в пікселах step_grid_y =
1.0f / N_step_grid_y; // значення кроку по осі y step_grid_y_pix = (Int)
(step_grid_y * M_y) // кроку по осі y в пікселах
// Вибираємо червоне перо товщиною 1:
Pen redPen = new Pen (Color.Red, 1);
// Будуємо в циклі горизонтальні лінії сітки від нульової лінії (осі x) вгору:
// Для цього визначаємо координати граничних точок сітки
int j_y; // лічильник для циклу
int y1_pix = oy_pix;
for (j_y = 1; j_y <= N_step_grid_y; j_y ++)
{
y1_pix = y1_pix - (int) step_grid_y_pix;
// Задаємо координати двох граничних точок лінії сітки: Point point3 =
new Point (ox_pix, y1_pix) // ліва крапка
Point point4 = new Point (x_point_end_pix, y1_pix) // права крапка
// Будуємо пряму лінію через дві заданий точки:
myGraphics.DrawLine (redPen, point3, point4)
}
// Будуємо ось ординат "oy" від y = 0 до y = 1;
// оголошую і задаємо ординату останньої точки осі ординат "y" при y = 1:
float y_point_end_pix;  y_point_end_pix =
oy_pix;
// Вибираємо зелене перо товщиною 2;
Pen greenPen = new Pen (Color.Green, 2);
// Задаємо координати двох граничних точок осі y:
Point point5 = new Point (ox_pix, indent_pix) // Верхн. точка в pictureBox
Point point6 = new Point (ox_pix, (int) y_point_end_pix) // нижня
// Будуємо лінію через дві заданий граничні точки:
myGraphics.DrawLine (greenPen, point5, point6)
// Будуємо вертикальні лінії сітки від осі y вправо
int  j_x;  float  x1;  int
x1_pix;
step_grid_x = (Float) (1.0f / N_step_grid_x) // крок град. знач. аргумен.
step_grid_x_pix = (int) (step_grid_x * M_x) // крок град. в пікселах x1_pix =
ox_pix;
for (j_x = 1; j_x <N_step_grid_x; j_x ++)
{
x1_pix = x1_pix + (int) step_grid_x_pix;
// Задаємо координати двох граничних точок лінії сітки:
Point point7 = new Point (x1_pix, indent_pix) // ліва крапка
Point point8 = new Point (x1_pix, (int) y_point_end_pix) // права
// Будуємо пряму лінію через дві заданий точки:
myGraphics.DrawLine (redPen, point7, point8)
}
```

```
// записуємо числа по осях координат, користуючись функцією DrawString (msg, ...):
// оголошую локальні змінні:
int n; float p1 = 1.0f; float p2; string msg;
// записуємо числа по осі "+ oy":
for (n = 0; n <= N_step_grid_y - 1, n ++) //  9
{
// p2 = p1 - n * 0.1F; ця формула виводить знач. у вигляді беск. дробу, тому
// множимо обчислене значення на 100.0f, округляємо його за допомогою
// функції Math.Round і ділимо результат на 100.0f
// p2 = (float) (Math.Round ((p1 - n * (float) 1 / N_step_grid_y) * 10.0F)) / 10.0F;
p2 = (Float) (Math.Round
        ((p1 - n * (Float) 1 / N_step_grid_y) * 100.0F)) / 100.0F; msg =
Math.Round  (p2  *  N_step_grid_y).ToString  ();  // Отримуємо значення
myGraphics.DrawString (msg, this.Font, Brushes.Blue,
ox_pix - 20 oy_pix - 215 + N * step_grid_y_pix)
}
// записуємо числа по осі "+ ox":
f loat kf = 0.00001f; // Коеф. для виведення в зручному вигляді
p2 = 0.0f; // Нач. значення для виведення нуля на початку координат
for (n = 1; n <= 7; n ++)
{
msg = p2.ToString ();
myGraphics.DrawString (msg, this.Font, Brushes.Blue, ox_pix -
55 + n * step_grid_x_pix, oy_pix + 5);
// Щоб без нескінченної дробу вивелися все значення по осі "ox",
// Помножити і ділити доводиться на 1000000.0f
p2 = (Float) (Math.Round ((0.1F * kf) * 1000000.0f)) / 1000000.0f; kf = kf *
10.0f;
} // ---------------------- Тепер будуємо графік ---------------------
-- ----------
```



Список рекомендованої літератури

1. Голуб Б.М. C#. Концепція та синтаксис. Навч. посібник. – Львів: Видавничий центр ЛНУ імені Івана Франка, 2006. – 136 с.
2. Троелсен Э. Язык программирования C# 5.0 и платформа .NET 4.5, 6-е изд.: Пер. с англ. – М.: ООО “И.Д. Вильямс”, 2013. – 1312 с.
3. Шилдт Г. C# 3.0. Полное руководство / Пер. с англ. – М.: Диалектика-Вильямс, 2009. – 992 с.
4. Котов, О.М. Язык C#: краткое описание и введение в технологии программирования: учебное пособие / О.М. Котов. – Екатеринбург: Изд-во Урал. ун-та, 2014. – 208 с.
5. Уотсон К. Microsoft Visual C# 2008. Базовый курс / К. Уотсон, К. Нейгел, Я.Х. Педерсен, Дж. Д. Рид, М. Скиннер, Э. Уайт. / Пер. с англ. – М.: Диалектика-Вильямс, 2009. – 1216 с.

Додаткова література:

1. Павловская Т.А. C#. Программирование на языке высокого уровня: Учебник для вузов. – СПб.: Питер, 2014. – 432 с.
2. Нейгел К., Ивьен Б., Глинн Д. C# 4.0 и платформа .NET 4 для профессионалов / Пер. с англ. – К.: Диалектика, 2011. – 1440 с.
3. Рихтер Дж. Программирование на платформе Microsoft .NET Framework 2.0 на языке C#. / Пер. с англ. – СПб: Питер, М: Русская Редакция, 2007. – 656 с.
4. Агапов В.П. Основы программирования на языке C#: учебное пособие / В. П. Агапов. – Москва: МГСУ, 2012. – 128 с.
5. Петцольдт Ч. Программирование для Microsoft Windows на C#. В 2-х томах. Том 1. /Пер. с англ. – М.: Издательско-торговый дом «Русская Редакция», 2002. – 576 с.
6. Андрианова А.А., Исмагилов Л.Н., Мухтарова Т.М. Объектно-ориентированное программирование на C#: Учебное пособие / А.А. Андрианова, Л.Н. Исмагилов, Т.М. Мухтарова. – Казань: Казанский (Приволжский) федеральный университет, 2012. – 134 с.
7. Вирт, Н. Алгоритмы и структуры данных [Текст]: пер. с англ. / Никлаус Вирт. – СПб: Невский Диалект, 2008. – 352 с.
8. Культин Н. Б. C# в задачах и примерах. – СПб.: БХВ-Петербург, 2007. – 240 с.
9. Ватсон К. C# / К. Ватсон, М. Беллиназо, О. Корне, Д. Эспиноза, З. Грин-фосс и др. / Пер. с англ. яз. – М.: Изд. «Лори». – 2005, 862 с.
10. Бокс Д., Селлз К. Основы платформы .NET, том 1. Общеязыковая исполняющая среда.: Пер. с англ. – М.: Издательский дом «Вильямс», 2003. – 288 с.
11. Либерти Дж., Программирование на C#. Создание .NET-приложений. Изд. 2-е. / Пер. с англ. – Спб.: Изд. «Символ», 2003. – 68